UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE GRADO

Visualización de redes de conceptos científico-técnicos

Daniel Santonja Merino

Tutor: Pablo Haya Coll

Julio 2014

Resumen

Los mapas de conceptos se han convertido en herramientas de aprendizaje muy potentes y el mundo educativo está empezando a utilizarlos. A pesar de ello no existe todavía aplicaciones que permitan aprovechar todo el potencial de esta herramienta.

Atendiendo a un hueco existente en ámbito de las herramientas software que existen dedicadas al uso de mapas de conceptos, se ha desarrollado una proyecto icación basada en dos pilares, una *API REST* y una serie de aplicaciones web que se alimentan de esa *API* .

El hecho de realizar una aplicación web nos permite llegar a todos los usuarios siempre que esto dispongan de una conexión a internet, favoreciendo así el desarrollo colaborativo de los mapas.

Para el desarrollo de este proyecto se han utilizado tecnologías punteras, como son bases de datos basadas en grafos. También se ha buscado una implementación que permita el aprovechamiento del código para ampliaciones y para su uso por parte de software de terceras personas. Para la parte del servidor se ha utilizado el lenguaje Python, y más concretamente el framework Django de aplicaciones web junto con la base de datos Neo4j, y para la parte del cliente se ha utilizado HTML5 y JavaScript.

El resultado de este trabajo es una aplicación robusta que permite la creación de mapas a partir de mapas externos o bien utilizando la *API* que se expone, al elegir una arquitectura *REST*, se ha apostado por un paradigma conocido y de éxito en el mundo del las tecnologías web.

Para la parte del cliente se ha desarrollado un módulo que permite la generación y visualización de grafos en el navegador web y la comunicación con la parte del servidor. La única aplicación finalizada es la que te permite navegar a través del mapa de conceptos, explorando las relaciones existentes entre los mismos.

Palabras clave: Mapas de conceptos, API REST, aprendizaje, Django, Javascript

Abstract

Concept maps have become powerful learning tools and the educational environment has started to use them. Despite this, there isn't yet any application that makes the most of the potential this tools have.

Taking notice of the existent gap in the software applications that currently exists focused on concept maps, it has been developed a project based on two pillars, a REST API and several web applications that feeds on the API

The fact of developing a web application allow s to reach every user, provided that the have internet access, regardless of the operative system they may use, favoring the collaborative develop of maps.

For the developing of this project, state of the art technologies have been used such as graph based databases. Also, it has been implement in such way that allows for the reuse of code both for code extensions and for 3rd-party software. On the server side Python has been the language of choice, most precisely the web framework Django plus the Neo4j database. On the client side the choices have been HTML5 and JavaScript.

The result of this project is a robust application that allows to create concept maps from other resources or using the API, by choosing a REST architecture, the bet is on a well known and successful paradigm in the field of web technologies.

On the client side there has been developed a module for generating and displaying graphs in the browser, and for communicating with the server side. The only finished application allows you to navigate through the map, exploring the relationships between the concepts.

Key words: Concept maps, API REST, learning, Django, Javascript

Índice

Índice de figuras

Índice de tablas

1. Introducción

1.1. Motivación

Los mapas de conceptos o mapas conceptuales son diagramas que representan relaciones entre conceptos. Son una herramienta usada por ingenieros, escritores y que está tomando fuerza en los procesos educativos, dada su eficacia en favorecer la creación de relaciones explicitas entre conceptos y mejorar el pensamiento crítico[1] [2].

Por un lado los alumnos puede aprender mejor si se le va enseñando las relaciones entre los conceptos que van apareciendo, como si de un grafo se tratara. Otra manera de aplicar los mapas de conceptos a la enseñanza es dejando que sea el alumno el que lo cree, pudiendo ver así en que puntos de la materia tiene un peor entendimiento.

Actualmente, y a pesar de existir varios productos software enfocados a la creación y manejo de mapas de conceptos, no hay una adopción mayoritaria de los mismos por parte de los educadores. Aunque la tecnología necesaria para poder utilizarlos se encuentra disponible de forma extendida. La mayoría de los estos productos son programas individuales que ofrecen una gran cantidad de opciones y una curva de aprendizaje elevada, restándole importancia al mapa en sí.

Otra de las motivaciones del proyecto es explorar la posibilidad de generar estos mapas de conceptos de forma dinámica y colaborativa, a través de una plataforma que permita una utilización directa usando la aplicación web o usando software de terceros, siendo usada la parte del servidor solamente como soporte de los datos.

Por eso se ha pensado en la creación de una aplicación, que ocupe un espacio por ahora olvidado por estas aplicaciones, la web, accesible, ubicua y familiar para todo el mundo.

1.2. Objetivos

- Creación de una aplicación web que permita visualizar mapas de conceptos. La parte más básica consiste en la visualización de mapas construidos *ad-hoc* para poder ser empleados como material didáctico.
- Interfaz sencilla e intuitiva. Uno de los objetivos es atraer a la comunidad educativa y a los estudiantes a que usen este tipo de aplicaciones por lo que no se les debe asustar con una complejidad excesiva. Al fin y al cabo se busca digitalizar el dibujar un esquema.
- Creación de una *API* que permita a la aplicación ser utilizada por terceros o por otro tipo de dispositivo. Esto permite que la aplicación web, no se viera limitada a una interfaz web, si no que podrías ser extendida a programas clásicos de escritorio o a dispositivos móviles.
- Utilización de nuevas tecnologías, siempre que estas tengan un valor para la aplicación. Una vez superado el escalón inicial de incorporar una nueva tecnología, nos permiten olvidarnos de los detalles de implementación para poder centrarnos en mejorar el producto.

- Integración de fuentes externas de datos para la creación de redes de conceptos. Esto permite generar automáticamente mapas sin tener que crearlos de forma manual.

1.3. Solución propuesta

La solución que se propone es una aplicación web instalable en un servidor con la mínima complejidad. Está aplicación ofrece una *API REST* que permite realizar peticiones *HTTP* al servidor y obtener los datos de la aplicación aunque sea desde un agente externo que no se haya desarrollado durante este trabajo, como podría ser una aplicación de un dispositivo móvil.

La aplicación web también ofrecerá unas páginas web, con la funcionalidad de comunicarse con el servidor ya implementadas. Estas páginas web permitirán navegar por los mapas de conceptos que existan así como en un futuro ofrecer funcionalidades nuevas sobre los susodichos mapas, como por ejemplo ejercicios de aprendizaje.

Cada usuario tendrá acceso a los mapas que él haya creado, los que quiera hacer públicos en el servidor así como un mapa de conceptos pre-cargado con la aplicación.

El escoger un formato de aplicación web permite desentendernos de los sistemas operativos del usuario, dado que con que tengan un navegador web podrán acceder a las funcionalidades del proyecto.

1.4. Organización de la memoria

1. **Introducción.** Se expone brevemente el sentido de este proyecto, así como los objetivos marcados
2. **Análisis de requisitos.** Se describe el problema de forma más concisa y se entra en detalle de que diferencia está aplicación del resto
3. **Diseño de la aplicación.** Se describe el estado del proyecto, su arquitectura y diseño seguido
4. **Desarrollo, pruebas y resultados.** Se explica como se han realizado cada uno de los pasos del desarrollo de una aplicación
5. **Conclusiones y trabajo futuro.** Se expone las conclusiones que se han obtenido tras el proceso y se hace un análisis del posible futuro de la aplicación

2. Análisis del problema

2.1. Estado del arte

2.1.1. ConceptNet

ConceptNet[1] es un proyecto que busca representar el conocimiento humano de forma que se aproxime a como se expresa en el lenguaje natural y se engloba en el *Common Sense Computing Initiative*. El objetivo final es recoger el conocimiento que un ordenador debería poseer para entender de que habla la gente.

La estructura subyacente de ConceptNet es un hipergrafo, en el que se encuentran palabras y frases comunes en el lenguaje escrito. Estas frases o palabras son denominados conceptos y cada uno de ellos representa un nodo. Para describir cada frase o palabra ConceptNet no tiene una definición, si no que el grafo en sí y las relaciones que tiene cada nodo con sus vecinos son lo que las definen. Las relaciones del grafo comprenden tanto relaciones más formales como la relación *IsA* para indicar una relación de subtipo, así como relaciones de sentido común"[5] como *LocatedNear* o *Desires*.

ConceptNet bebe de numerosas fuentes, y va incrementando su tamaño en cada iteración, la versión utilizada y más actual es *conceptnet5*, y usa de fuentes: la página web de *Open Mind Common Sense*[2], *games with a purpose*, el wikidiccionario en inglés[3], *WordNet 3.0*[4], *DBPedia*[5] y *Wikipedia*[6] e incluye conceptos en varios idiomas como ingles, chino o portugués.

2.1.2. Bases de datos no relacionales

Las bases de datos no relacionales, también llamadas bases de datos NoSQL, son bases de datos que se alejan del modelo tradicional de base de datos formado a partir de datos y relaciones tabuladas. La primera vez que se usó el término NoSQL fue en 1998 por Carlo Strozzi aunque no hubo una explosión de bases de datos no relacionales hasta mediados de los 2000 con *Hadoop*[7], *Dynamo*[8], *BigTable*[9] y *MongoDB*[10] entre otros.

La motivación para la existencia de estas bases de datos es muy variada, algunas buscan escalabilidad para que el rendimiento de la base de datos no se reduzca considerablemente cuando el numero de datos aumenta de forma notable, otras buscan almacenar las entidades de forma que cierto tipo de consultas sean más rápidas que con una base de datos relacional mientras que otras simplemente apuestan por manejar tipos de datos que no tienen sentido que sean modelados en forma de tabla.

[1] `http://conceptnet5.media.mit.edu`
[2] `http://openmind.media.mit.edu`
[3] `http://en.wiktionary.org`
[4] `http://wordnet.princeton.edu`
[5] `http://dbpedia.org/About`
[6] `http://www.wikipedia.org`
[7] `http://hadoop.apache.org/`
[8] `http://en.wikipedia.org/wiki/Dynamo\_(storage\_system)`
[9] `http://en.wikipedia.org/wiki/BigTable`
[10] `http://www.mongodb.org/`

Hay varias maneras de clasificar las bases de datos no relacionales, pero estas clasificaciones tienden a solaparse. En la tabla 1 vemos una comparación de distintos tipos de bases de datos en función del modelo de almacenaje que utiliza.

Modelo de dato	Rendimiento	Escalabilidad	Flexibilidad	Complejidad	Funcionalidad
Almacenamiento clave-valor	Alto	Alto	Alto	Ninguno	Variable (ninguno)
Orientado a columnas	Alto	Alto	Moderado	Bajo	Mínimo
Orientado a documentos	Alto	Variable (Alto)	Alto	Bajo	Variable (bajo)
basada en grafos	Variable	Variable	Alto	Alto	Teoría de grafos
Base de datos relacional	Variable	Variable	Bajo	Moderado	Álgebra relacional

Tabla 1: Comparativa de bases de datos. [3]

2.2. Requisitos de la solución

Los principales requisitos de la aplicación son;

- Modelar los mapas de conceptos en una base de datos
- Ofrecer un punto de acceso a la base de datos a través de una *API REST*.

  Posibilidad de crear nuevos conceptos de forma remota.

  Posibilidad de modificar conceptos creados de forma remota.

  Posibilidad de acceder a esos conceptos a traves de identificadores únicos.

  Posibilidad de eliminar conceptos a traves de los identificadores.

  Posibilidad de crear relaciones entre conceptos.

  Posibilidad de eliminar las relaciones entre conceptos.

- Ofrecer un proyecto web que permita usar la API desarrollada para ofrecer sobre ella funcionalidades avanzadas

  Página web que permita navegar por un mapa.

  Página web que permita crear y modificar mapas.

  Página web que permita relaizar ejercicios didacticos sobre los mapas.

2.3. Conclusiones

Tras el análisis de las tecnologías punteras en este campo que existen hoy en día, parece que se adecuan a nuestros requisitos, dentro de las bases no relacionales surge de forma casi inmediata la relación entre las bases de datos basadas en grafos y los mapas de conceptos.

Por otro lado Conceptnet nos permite disponer de una base de datos ya creada para poder usarla tanto para pruebas como para la aplicación final, es un gran mapa de conceptos que encaja a la perfección con los datos necesarios para la aplicación.

3. Diseño de la aplicación

3.1. Análisis de las tecnologías

3.1.1. Neo4j

Neo4j[11] es una base de datos basada en grafos, implementada en Java y de código abierto. Creada y distribuida por Neo Technology, Neo4j se distingue de las bases de datos tradicionales en que la estructura subyacente con la que trabaja el programa y la que se expone al usuario es un grafo a diferencia del modelo de tablas de las bases de datos relaciones.

Neo4j se crea con el objetivo de abordar la creciente complejidad de los diseños de las bases de datos, que las bases de datos tradicionales no son capaces de manejar sin un impacto profundo en su rendimiento, entidades altamente relacionadas dan lugar a consultas complejas y de alto coste computacional. Asimismo, al trabajar con relaciones no reciprocas y con niveles de profundidad elevados en las consultas debido al uso de *joins* recursivos, como se puede observar en la tabla 2.

Profundidad	Tiempo de ejecución en BD (s)	Tienpo de ejecución en Neo4j(s)	Registros devueltos
2	0.016	0.01	2500
3	30.267	0.168	11000
4	1543.505	1.359	600000
5	Sin terminar	2.132	800000

Tabla 2: Comparación de tiempos para la misma consulta en una base de datos relacional y en Neo4j. [4]

El hecho de que Neo4j esté basado en grafos implica que está diseñado y optimizado para trabajar con relaciones, lo que le permite manejar un grado de complejidad de diseño muy alto. Ademas Neo4j es semi-estructurado, eso significa que permite un número ilimitado de tipos de relaciones y de entidades o nodos, que la estructura de la base de datos no es inalterable. Las modificaciones al esquema de la misma son triviales, así como la adición de atributos a los nodos. Debido a su estructura de grafo y a que carece de un nodo central o raíz, para poder encontrar un nodo determinado sin tener que recurrir al identificador numérico asignado automáticamente, Neo4j permite la creación de índices sobre ciertos atributos para mejorar el rendimiento de las búsquedas.

Cypher es el lenguaje de consulta utilizado en Neo4j, es un lenguaje muy visual, donde las condiciones de la consulta se pueden ver ademas de utilizar comandos habituales en *SQL* tales como *WHERE, ORDER BY o WITH*. Un ejemplo de una consulta sencilla sería:

```
START n=node({id})
MATCH (n)-[:FRIEND]-(friend)-[:LIKES]-(item)
WHERE item.rating > 5
RETURN DISTINCT item
```

[11] `http://www.neo4j.org`

START indica el nodo desde el que se quiere empezar la consulta, para partir desde todos los nodos, bastaría con usar n=node(*) pero para grafos grandes no se recomienda su uso. MATCH es el comando mas distintivo de *Cypher* y permite especificar el camino de la consulta, discriminado por tipo de relación [:FRIEND] y declarando variables (item) para ser usadas mas adelante en la consulta. WHERE cumple la misma función que en lenguajes SQL, discriminar por el contenido de los datos, aquí se puede acceder a las variables declaradas en la clausula MATCH. Por último RETURN determina los valores de retorno de la consulta y DISTINCT hace que no haya duplicados. En este caso la consulta devuelve aquellos objetos que les han gustado a los amigos del usuario y que tienen una puntuación mayor de 5. Con una consulta, *Cypher* nos ha permitido crear un sistema de recomendación rudimentario.

3.1.2. Django

Django es un *framework* de desarrollo web implementado en Python y de código abierto. Aunque nacido para crear y gestionar páginas de noticias, ha evolucionado para ofrecer una manera rápida y sencillas de crear sitios web complejos. Django sigue el patrón modelo-vista-controlador (MVC), que en su caso corresponden a las partes *model*, *template* y *view* respectivamente. Los modelos son clases de Python que después se replican automáticamente en la base de datos con la que se va a trabajar. Este framework ofrece una API para realizar consultas a la base de datos a través de los modelos definidos. Los módulos *views* ofrecen la funcionalidad de un controlador, realizando las llamadas necesarias a los modelos y generalmente contienen la lógica de negocio. Por último los *templates* o plantillas permiten la generación dinámica de contenido html.

A la hora de organizar un proyecto la estructura que utiliza Django es una estructura basada en aplicaciones. Un proyecto está conformado por aplicaciones que, de haber sido desarrolladas de forma correcta, pueden ser reutilizadas en cualquier otro proyecto. Cada aplicación gestiona el direccionamiento url interno de la aplicación, define los modelos que necesita y las plantillas necesarias para la creación de las páginas web. Siendo el proyecto el vínculo cohesionador que determina la configuración y el direccionamiento urls en su nivel más alto.

Uno de las características más potentes de Django es su sistema de plantillas. Por un lado permite la generación de código html estático como cualquier framework, pero ademas tiene un *SDL*(Specific-domain language) propio con secuencias de control, manejo de cadenas con un énfasis especial en los problemas típicamente asociados al desarrollo web y la posibilidad de ser ampliado con funciones propias. La complejidad del lenguaje es tal que tiene algunas características de lenguaje orientado a objetos como composición o herencia de plantillas.

Para la creación de la *API REST*, se ha usado el modulo de Django Tastypie. Tastypie permite crear de forma semiautomática una *API REST* dado un modelo de Django. Para bases de datos relacionales está creación es trivial y no implica más que instalar el modulo y declarar sobre que modelos se quiere crear la *API*. Como en este caso hemos usado Neo4j, está aproximación no ha sido posible y se ha usado las herramientas que Tastypie pone al servicio del desarrollador para personalizar el funcionamiento interno del servicio.

Para la interacción con la base de datos se ha usado *neo4django*. El objetivo de este modulo es ofrecer la misma funcionalidad que da Django con sus modelos y las bases de datos tradicionales para la base de datos Neo4j. Cabe destacar que todavía se encuentra en desarrollo y es el modulo que más limita al resto de herramientas en cuanto a versiones compatibles.

3.1.3. JavaScript

JavaScript es un lenguaje de programación orientado al desarrollo web en el lado del cliente. Este lenguaje de tipado dinámico, basado en prototipo que soporta la programación orientado a objetos. Fue diseñado con la idea en mente de un lenguaje parecido a Java pero más ligero que pudiera funcionar en el navegador *Nestcape*.

Una de las bibliotecas más extendidas y utilizadas de JavaScript es jQuery. Nació cuando el manejo y modificación del DOM por parte de JavaScript nativo era complicado y cada navegador lo implementaba de forma diferente, permitiendo simplificar considerablemente estas tareas Ademas jQuery ofrece soporte multi-navegador, asegurando el mismo comportamiento en todos los navegadores soportados.

La biblioteca de JavaScript en la que se basa la parte del cliente del proyecto es la librería arbor.js. Esta biblioteca permite el manejo y el calculo de coordenadas de un grafo, esto es, permite crear la estructura de un grafo en JavaScript y calcular la disposición de los nodos en función de los pesos de los mismos y de otros parámetros.

3.1.4. REST y servicios RESTful

REST significa *Representational State Transfer*, y es una técnica de arquitectura software orientada a la comunicación entre maquinas. REST define unas guías de arquitectura y unas propiedades que emergen de esas guías o restricciones, no entra en detalles de implementación, dejando a cada cual que la realice como prefiera. *REST* fue definido por Roy Fielding en su tesis doctoral en el año 2000 [7]. Cuando un servicio web o *API* implementa *REST* se dice que es *RESTful*.

Ahora pasaremos a describir las restricciones que impone *REST*, entraremos en menor o mayor grado según el impacto de la restricción en la aplicación.

- **Cliente-Servidor:**Tiene que haber una separación clara de responsabilidades entre las dos entidades, el cliente no tiene que encargarse de el almacenamiento de datos, pero es el encargado de pedir la información, por otro lado el servidor no entiende de estados del cliente ni de la representación visual de los datos, esto permite a ambos componentes evolucionar de forma separada

- **Sin estado:** Toda comunicación debe asumir que no se está manteniendo un estado entre mensajes, así cada mensaje debe incluir toda la información necesaria en cada petición. Esta restricción mejora ciertas propiedades del sistema que esté implementando *REST*: Visibilidad; fiabilidad, pues es fácil recuperarse de un fallo al no haber perdida de estado; y escalabilidad, al no tener que almacenar estados entre peticiones y poder liberar recursos de forma inmediata. Por contra el no mantener

un estado puede incrementar la congestión de la red al tener que enviar todo el rato información repetida debido a que el servidor no guarda ninguna información para garantizar el contexto.

- **Caché:** Para mejorar la eficiencia de la red se añade la restricción de la caché. Dicha restricción requiere que el contenido que sirve el servidor sea marcado como cacheable o no cacheable, si una respuesta del servidor es cacheable el cliente tiene el derecho y la oportunidad de reutilizar ese contenido más adelante. Con esta reutilización el cliente puede eliminar algunas de las interacciones con el servidor, peor a cambio, se puede producir cierta inconsistencia entre la información que mantiene el cliente y la que pueda tener el servidor.

- **Interfaz uniforme:** El restricción de la interfaz uniforme es una de las características clave de una arquitectura REST. Aplicando el principio de ingeniería del software de generalizar a la interfaz, se consigue que la arquitectura global del sistema se simplifique. Ademas la implementación queda separada de los servicios ofrecidos, lo cual permite que ambos evolucionen de forma independiente. El punto negativo es que al generalizar tanto, se pierde en eficiencia, pues en vez de ofrecer una solución especifica para cada problema se tiene que usar una forma estandarizada de transferir la información

- **Sistema de capas:** Para mejorar la escalabilidad se requiere un sistema de capas jerarquizado en el que cada capa no puede ver más allá de la capa inmediatamente contigua. Una vez más esta restricción aumenta la independencia entre los componentes del sistema. Así una capa puede colocarse encima de un *legacy system* o proteger a los nuevos servicios de clientes antiguos. Que existan capas intermedias favorece la escalabilidad, pues permite balancear la carga en más niveles. La contraparte negativa es que puede disminuir el rendimiento, al aumentar el numero de llamadas del sistema, pero si se utiliza inteligentemente el sistema de caches del que se habló anteriormente, este defecto puede verse compensado.

- **Código bajo demanda:** Esta restricción que es la única opcional, permitiría a los clientes descargar código desde el servidor y ejecutarlo. Esto puede permitir simplificar el cliente, que puede ser implementado rápidamente y utilizar el código que provee el servidor hasta que el total de sus funcionalidades estén implementadas en el cliente. Por otro lado esta restricción disminuye la visibilidad y la seguridad del sistema, haciendo que la mayoría de las veces sea descartada para ser incorporada en la arquitectura del sistema.

Aquí definiremos de forma breve como se produce esa comunicación entre maquinas, para un servicio *REST* existen recursos, estos está identificados en sus sistema por un identificador conocido como URI, que en el caso de Internet se correspondería con una *URL*, una vez localizado ese identificador, el servidor responderá con una representación del recurso, hay que tener en cuenta que un recurso puede tener varias representaciones, dependiendo del cliente que este realizando la petición o de los parámetros de la misma, así pues a través de una *URL* se puede obtener tanto un archivo *JSON* como un archivo *HTML* listo para ser renderizado en un navegador.

Los recursos de un servicio *REST* pueden ser considerados como los sustantivos de esta arquitectura, para operar sobre ellos se tienen los comando *HTTP*: *GET, PUT, POST* y *DELETE*.

- **GET:** Este comando nos permite obtener un recurso, es un comando que no debería modificar bajo ningún concepto el almacenamiento de los recursos que se están utilizando. Si el comando *GET* se utiliza sobre acompañado de una *URI* que apunte a un recurso, el servidor deberá responder con la representación oportuna de ese recurso. Si la *URI* no especifica un id, se devuelve una lista de recursos, el contenido de esa lista dependerá de los parámetros de la petición, así como de la implementación particular de cada servicio.

- **PUT:** Este comando permite modificar un recurso almacenado, si ese recurso no existiera se crearía. *PUT* debe ir siempre especificando un recurso en concreto. Además *PUT* es idempotente, esto es, realizar varias veces la misma petición *PUT* con los mismos parámetros debería dar el siempre el mismo resultado, esto en la practica quiere decir que no se crean recursos nuevos, si se ha aplicado un *PUT*, aplicar el mismo comando otra vez no alteraría la base de datos.

- **POST:** Este comando, parecido a *PUT*, permite l creación de nuevos recursos cuando no se dispone del identificador que corresponde a ese recurso, bien porque son identificadores auto generado y gestionados por el servicio, bien porque no se dispone de esa información. Un comando *POST* no debe indicar el identificador del recurso a crear, y la respuesta del servicio debe ser precisamente la *URI* que identifica al nuevo recurso.

- **DELETE:** *DELETE* es la contrapartida de *PUT*, este comando elimina el recurso especificado, y al igual que *GET* está permitido su uso sin indicar el identificador de un recurso, pasando a estar definido su comportamiento por la implementación particular del servicio.

3.2. Arquitectura del sistema

El esquema de la aplicación se puede observar en la figura 1

- **Conceptnet(1):** El hipergrafo de ConceptNet.

- **toNeo4j (2):** Es un *script* que extrae la información del hipergrafo para insertarlo en la base de datos Neo4j de forma que los datos puedan ser utilizados por la aplicación.

- **Neo4j(3):** La base de datos Neo4j, que almacena la red de conceptos.

- **neo4django(4):** El módulo de Django que permite usar los modelos de Django sobre Neo4j para poder usar el sistema de consultas y automatizar la creación de los nodos clase.

- **La *API* de Neo4j (5):** Se usa cuando se necesitan realizar consultas complejas, dado que permite aprovechar toda la expresividad del lenguaje Cypher de Neo4j, ademas, al eliminar una capa intermedia de procesamiento, aumenta la velocidad de las consultas.

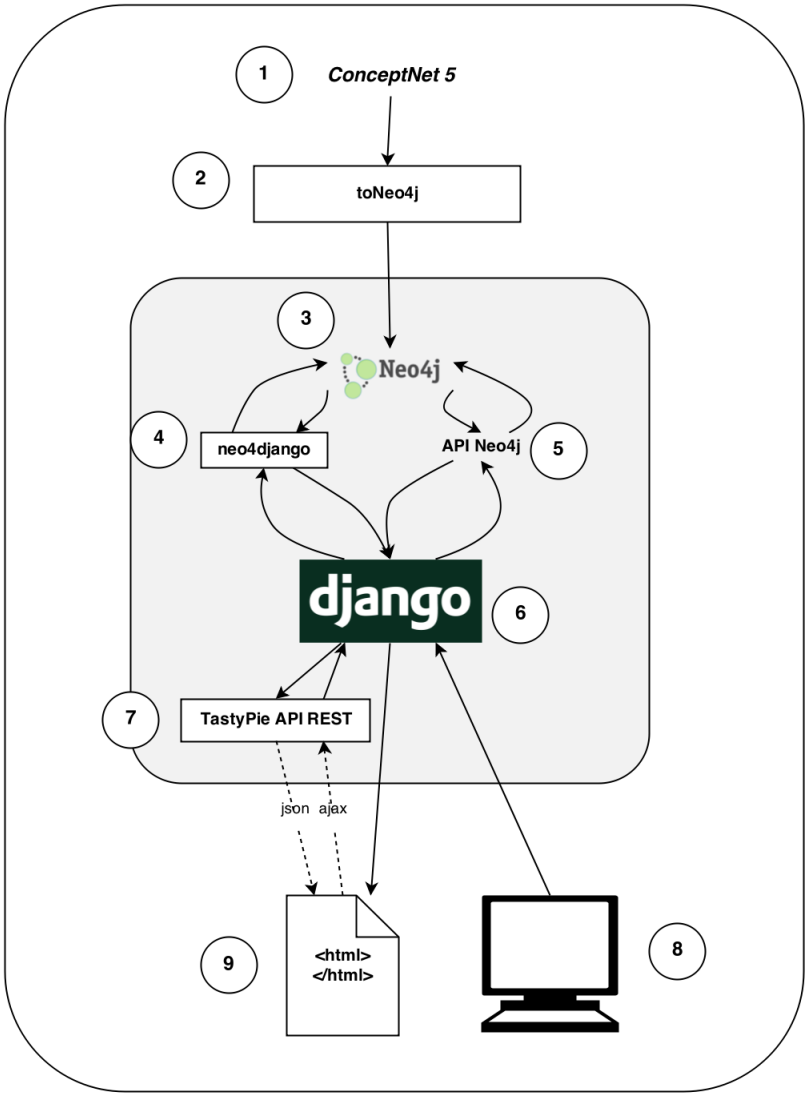

Figura 1: Arquitectura de la aplicación

- **Django (6):** Es el núcleo de la aplicación. Se encarga de resolver las peticiones *HTTP* que llegan de los clientes, distinguir si las mismas van dirigidas a la *API* o a obtener una página web y llamar a la vista correspondiente. Sirve de enlace entre todos los demás elementos de la aplicación.

- **TastyPie API REST (7):** Es el modulo encargado de la *API REST*, Django delega en él todo lo referente a este tema, gestiona los siguientes tipos de comandos *HTTP*: *GET POST PUT y* DELETE.

- **Cliente (8):** El usuario que a través de su navegador realiza una petición HTTP para cargar la página web de la aplicación.

- **La página web (9):** Presenta la aplicación al usuario y realiza peticiones *AJAX* a la *API* de la aplicación para servir los datos según va interaccionando el usuario.

3.3. Lado del servidor

3.3.1. Descripción estructural

La parte codificada del lado del servidor, y que conforma el núcleo del proyecto es la aplicación implementada. Esta se puede diferenciar en dos partes: por un lado la implementación de la *API REST*, destinada a dar un servicio trasparente a cualquier cliente. Por otro lado la implementación de una página web que sirve de navegador de los datos y que hace uso de dicha *API*.

Dada la estructura de Django y dado que Python no exige que toda la funcionalidad de un proyecto se encuentre encapsulada en clases, en el proyecto se pueden distinguir las secciones que si que implementan clases y las que no.

Ficheros cuya funcionalidad no venga encapsulada en una clase son:

- **urls.py:** Contiene la funcionalidad de direccionamiento de las peticiones, en caso de estar dirigidas a la *API*, delega en esta.
- **views.py:** Contiene los controladores de la aplicación, teniendo un controlador por página web ofrecida.

Ficheros con una estructura orientada a objetos:

- **models.py:** Define los modelos que se implementan luego en la base de datos, estos son los objetos básicos que se manejan en la aplicación, dado que representan nodos, heredan de la clase de neo4django NodeModel.

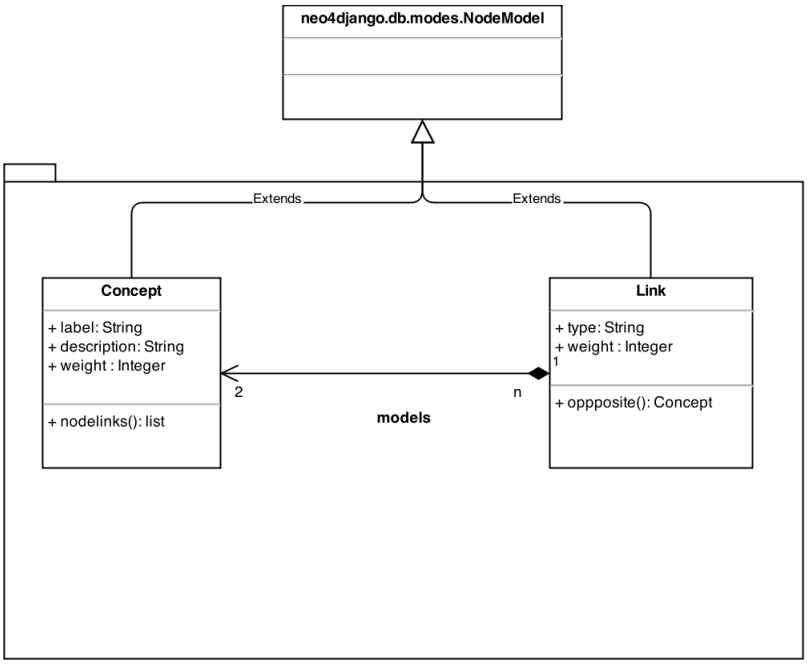

Figura 2: Paquete models

- **api.py:** Este modulo presenta la parte más compleja de la implementación, este modulo provee la funcionalidad *REST* de la aplicación, cada una de las dos clases que conforman este módulo modelan el comportamiento de los modelos de la base de datos ante las peticiones *HTTP*. Normalmente en *tastypie* se hereda de la clase *ModelResource*, que provee de las funcionalidades básicas por defecto. Pero esta clase solo funciona con bases de datos relacionales, lo que nos obliga a heredar de la clase *Resource* que solo ofrece el esqueleto a partir el cual construir la aplicación. Así pues estas clases contienen los métodos que son necesarios sobrescribir para que la el modulo *tastypie* funcione.

- **serializers.py:** Clases que sobrescriben el comportamiento por defecto de la transformación de objeto Python a cadena *JSON*, este modulo conforma junto con *api.py* el paquete *api*.

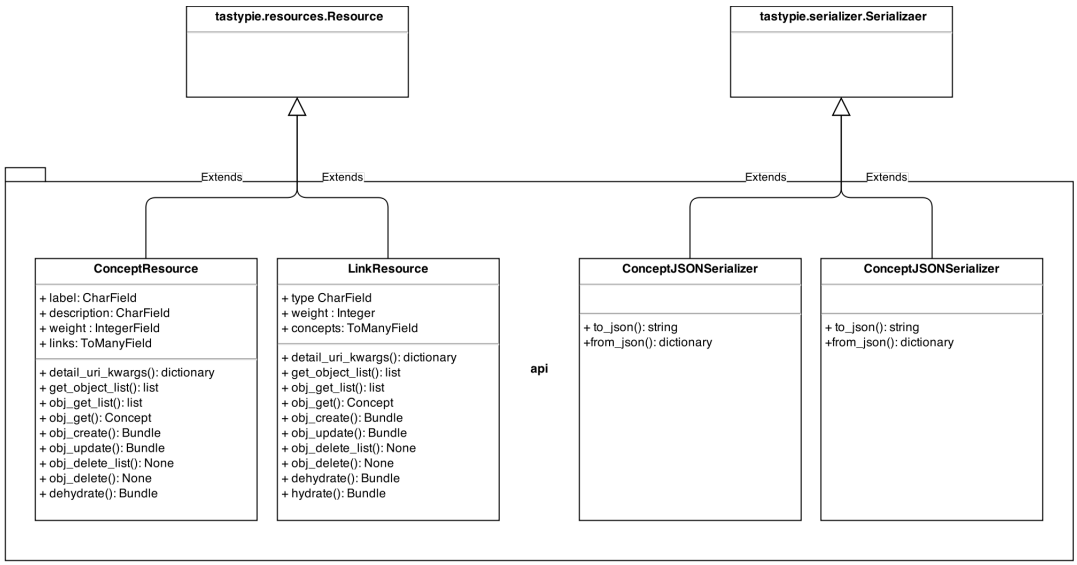

Figura 3: Paquete api

3.3.2. Descripción comportamental

Para mostrar el comportamiento de la aplicación nos centraremos en el módulo api.py. En la figura 4 se puede observar la secuencia de llamadas para un petición GET de tipo *get all* (p. ej. api/v1/concepts/). Este diagrama muestra el recorrido de llamadas que realiza *tastypie* de forma predeterminada cuando se usa con una base relacional típica. La implementación queda bien dividida en las distintas partes. La rama de la izquierda es la encargada de pedir los objetos a la base de datos aplicando los filtros que sean necesarios y la rama de la derecha aplica los permisos de los usuarios en caso de que estos existieran. Este diagrama de llamadas os ha servido de ejemplo para desarrollar el nuestro.

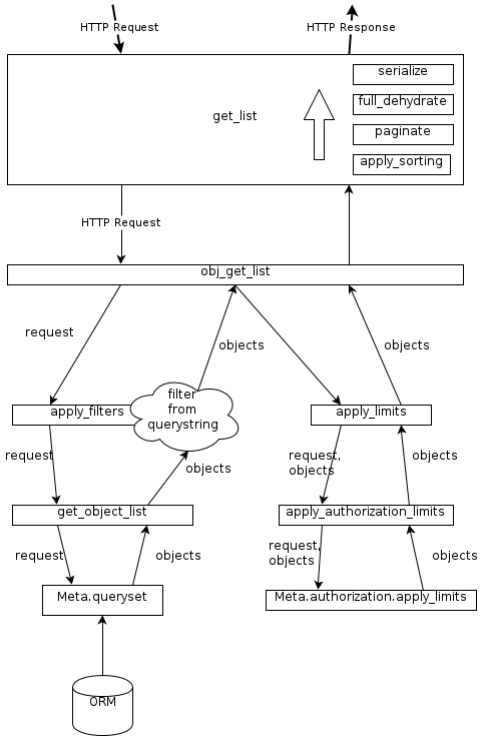

Figura 4: Ejemplo típico de una llamada la *API* [6]

En la figura 5 observamos un ejemplo sencillo de las transiciones que sufre la clase *ConceptResource*. Esta clase, nombrada así por seguir las convenciones de tastypie se encarga de modelar el comportamientos *REST* de la clase *Concept*. Es la que controla los parámetros de la petición *HTTP* y se encarga de hacer las consultas a la base *Neo4j*

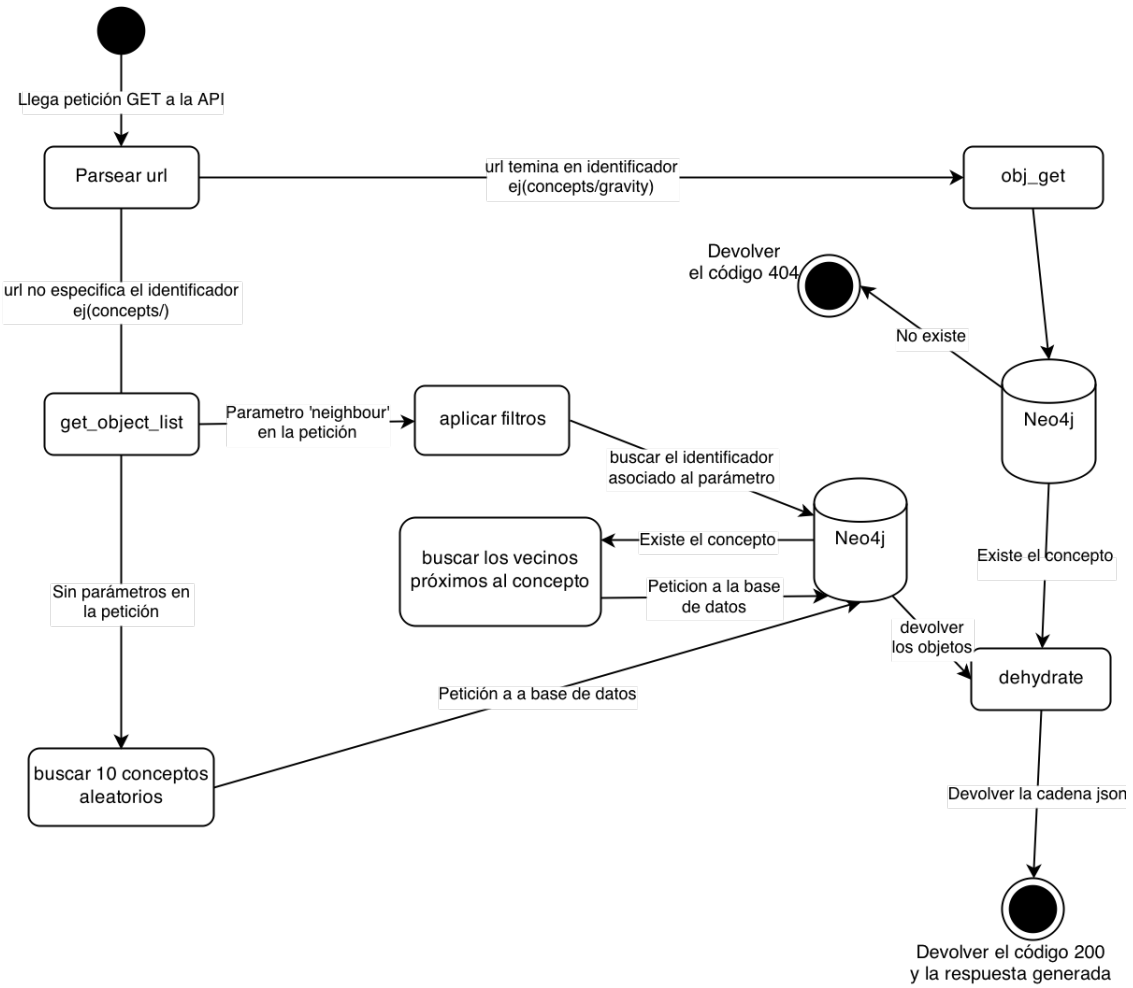

Figura 5: Transición de la capa subyacente de la *API*

Como se puede observar, los condicionantes para pasar a un estado a otro son el tipo de url de la petición, una url contiene el tipo de recurso que se quiere obtener y luego puede tener el identificador de un recurso en concreto, que en este caso es la etiqueta del concepto, lo cual hará que se devuelva el objeto en cuestión o un error 404 en caso de no encontrarse dicho recurso. En caso de no especificar el identificador del recurso que se quiere obtener se devuelve una lista de objetos, normalmente es en este tipo de llamadas en la que entran en juego los parámetros de la petición. Dada la limitación que sufre un servicio *RESTful* en cuanto al contenido de las *url* que puede usar, añadir nueva funcionalidad implica siempre usar parámetros de los comandos *HTTP*, en este caso si se encuentra el parámetro *neighbour* el servicio devolverá una lista con el recurso indicado y sus vecinos. Estas transiciones se repiten de forma más simple en el resto de comandos HTTP: *PUT, POST* y *DELETE*.

3.4. Lado del cliente

3.4.1. Descripción estructural

Como se ha descrito en el apartado 3.1.3, se ha utilizado la biblioteca arbor.js para el desarrollo de este proyecto, dicha librería provee de un sistema de manejo de los nodos y enlaces de un grafo, así como de la funcionalidad del cálculo de posiciones de los mismos. Esta biblioteca ofrece 1 clase que es la encargada de el manejo del grafo, *ParticleSystem*, así como 3 estructuras de datos que serán las encargadas de modelar nuestro mapa de conceptos: *Node*, *Edge* y *Point*. Aprovecharemos que la estructura *Node* puede almacenar un número arbitrario de atributos para añadir a los que ya existen los atributos *root* y *base*, que representan si un nodo es el nodo central, y el tamaño del nodo a dibujar respectivamente.

Para el desarrollo de esta aplicación se han creado dos clases que permiten realizar la peticiones al servidor, y encargarse de el renderizado del mapa. La figura 6 muestra estas dos clases, las funciones de incialización, *redraw* e *initMouseHandling* de *Renderer* vienen obligadas por *SystemParticle* dado que es a clase que hará uso de la nuestra. El resto de métodos sirven para ofrecer mayor modularidad, de cara a una ampliación de la clase más adelante. *Proxy* es la clase encargada de comunicarse con el servidor, y se basa en el sistema de *callbacks* típico de jQuery.

Cada página deberá desarrollar sus propias funciones que determinen el comportamiento del mapa al interaccionar con él. Estas funciones se pasan a las clases superiores en forma de *callbacks* que se asignaran a los métodos correspondientes.

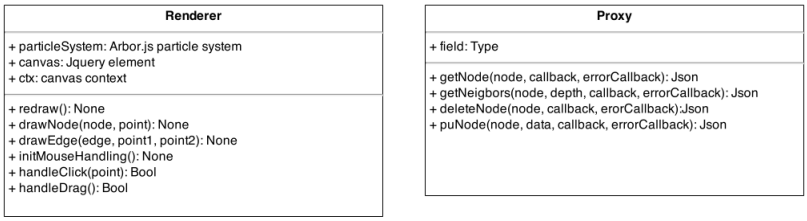

Figura 6: Clases del cliente

3.4.2. Descripción comportamental

En el caso del cliente, el modelo que se utiliza para realizar el renderizado de los nodos es un bucle infinito que va llamando secuencialmente a las funciones de calculo de posiciones y de renderizado, en este caso la librería arbor.js se encarga de manejar el subgrafo que hemos cargado del servidor y calcular las nuevas posiciones de los nodos si es necesario, habiendo sido implementado par el cliente la parte del renderizado y del manejo de eventos para navegar por el grafo.

Se adjunta la figura 7 que muestra el diagrama de transición de estados de la aplicación JavaScript.

Una vez que la página ha cargado, se deberá elegir un termino a partir del cual navegar, una vez elegido, la página realizará una petición *AJAX* a la aplicación web, si esta devuelve un valor válido se mostrará el concepto elegido y los conceptos con los que está relacionados. Si se pincha sobre uno de los términos relacionados, se repite el ciclo, pasando a ser considerado el concepto pinchado como el concepto central, y se obtendrá sus conceptos relacionados.

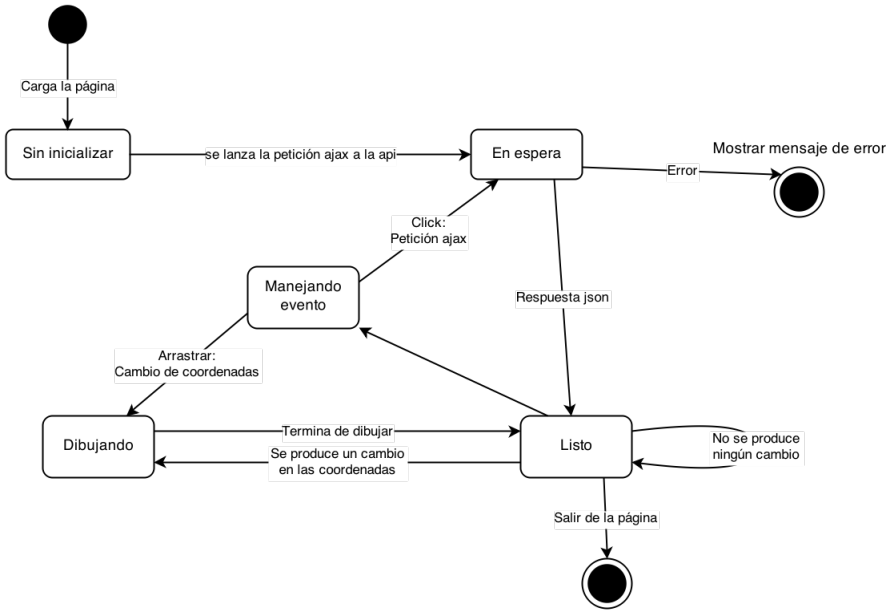

Figura 7: Transición de estados en el cliente

3.5. Diseño de la interacción de usuario

Cuando hablamos de usuarios de la aplicación, cabe hacer una distinción entre el usuario básico, que utilizará la aplicación a través de la página web y el usuario avanzado, que requerirá de conocimiento en desarrollo, que puede aprovechar todo el potencial de la aplicación a través de la *API*.

En esta sección se mostrará tanto el diseño de baja fidelidad inicial como el resultado fina con las pantallas de la aplicaciós, así como la descripción de la *API REST*

3.5.1. Descripción de la *API*.

La *API* expone dos recursos, los conceptos y los enlaces, para cada uno de ellos se ha desarrollado el contenido básico y necesario de un aplicación *RESTful*. Esto quiere decir los metodos *GET, POST, PUT* y *DELETE*.

A continuación pasan a describirse los servicios de los que puede aprovecharse un usuario avanzado.

Concepts Los conceptos son la pieza clave de la aplicación, son los recursos que más funcionalidad tienen dedicada:

- **GET concepts/:** Devuelve una lista de 10 conceptos aleatorios de la base de datos.
- **GET concepts/?neigbour=id&depth=x:** Los vecinos con profundidad x del concepto *id*, el parámetro *depth*es opcional, tomándose como valor por defecto una profundidad de 1.
- **GET concepts/?search=id:** Realiza una búsqueda no-exacta del termino indicado por *id*, devolviendo una lista con los posibles candidatos, hay que tener en cuenta que realiza una búsqueda tipo *LIKE*, es decir dado el la id *for*, devolverá *for*, *forensic*, *formula*, etc
- **GET concepts/id:** Devuelve el concepto con el identificador especificado, el identificador debe ser exacto, aunque el sistema es insensible a mayúsculas/minúscula.s
- **PUT concepts/id:** Crea el concepto *id* o bien lo actualiza si ya existiera.
- **POST concepts/:** Crea un concepto con los parámetros indicados, el servidor devuelve el *URI* del recurso.
- **DELETE concepts/:** Elimina todos los conceptos existentes.
- **DELETE concepts/id:** Elimina el concepto especificado.

Links Las relaciones entre conceptos también son recursos que pueden consultarse y crearse:

- **GET links/:** Devuelve una lista de 10 enlaces aleatorios de la base de datos.
- **GET links/?type=type:** Devuelve una lista con los enlaces del tipo *type*.
- **GET links/id:** Devuelve el enlace con el identificador indicado.
- **PUT links/id:** Crea el enlace *id* o bien lo actualiza si ya existiera.
- **POST links/:** Crea un enlace con los parámetros indicados, el servidor devuelve el *URI* del recurso.
- **DELETE links/:** Elimina todos los enlaces existentes.
- **DELETE links/id:** Elimina el enlace especificado.

3.5.2. Prototipo de baja fidelidad: *wireframes* de las pantallas de la aplicación

Para los *wire*frames se ha utilizado la herramienta *mockingbird*. La aplicación en un inicio constaba de 3 pantallas:

- **Index:** Página de inicio, interfaz de usuario minimalista que lleva a las otras pantallas.

Figura 8: *Wireframe* del indice

- **Navigate:** Busca un concepto y navega a partir de ese a otros a través de la relaciones.

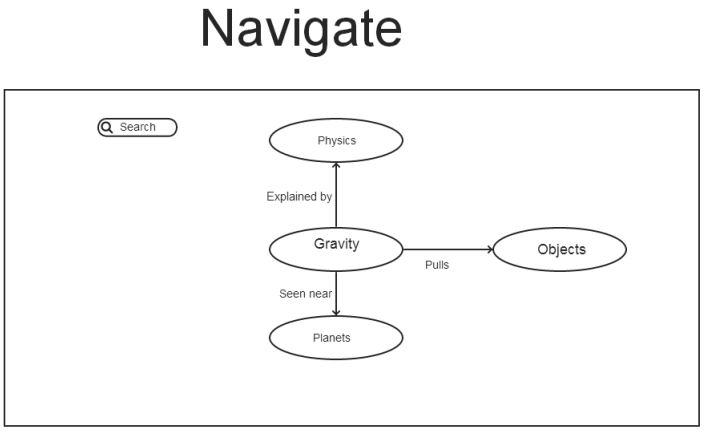

Figura 9: *Wireframe* de la pantalla de navegación

- **Learn:** Se muestra un concepto central y hay que rellenar datos que faltan

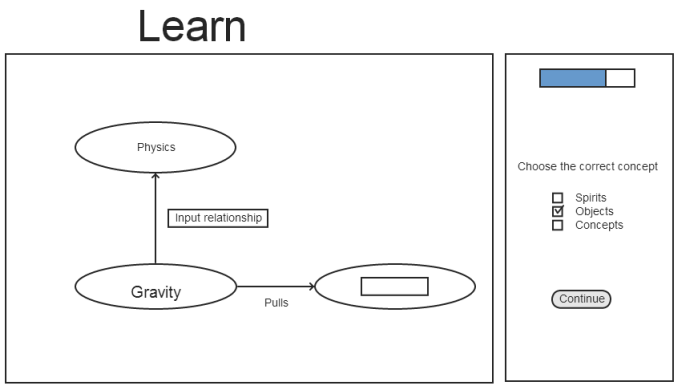

Figura 10: *Wireframe* de la pantalla de aprendizaje

3.5.3. Descripción de la aplicación desde el punto de vista del usuario, incluyendo capturas de pantallas.

La figura 11 muestra la pantalla de inicio de la aplicación, se ha optado por un diseño simple y minimalista, dado que el diseño gráfico no forma parte del contenido del trabajo. En la página se muestra un mensaje de bienvenida y dos botones que corresponderían a dos de las funcionalidades de la aplicación. Actualemente solo se encuentra desarrollada la funcionalidad de *navigate*.

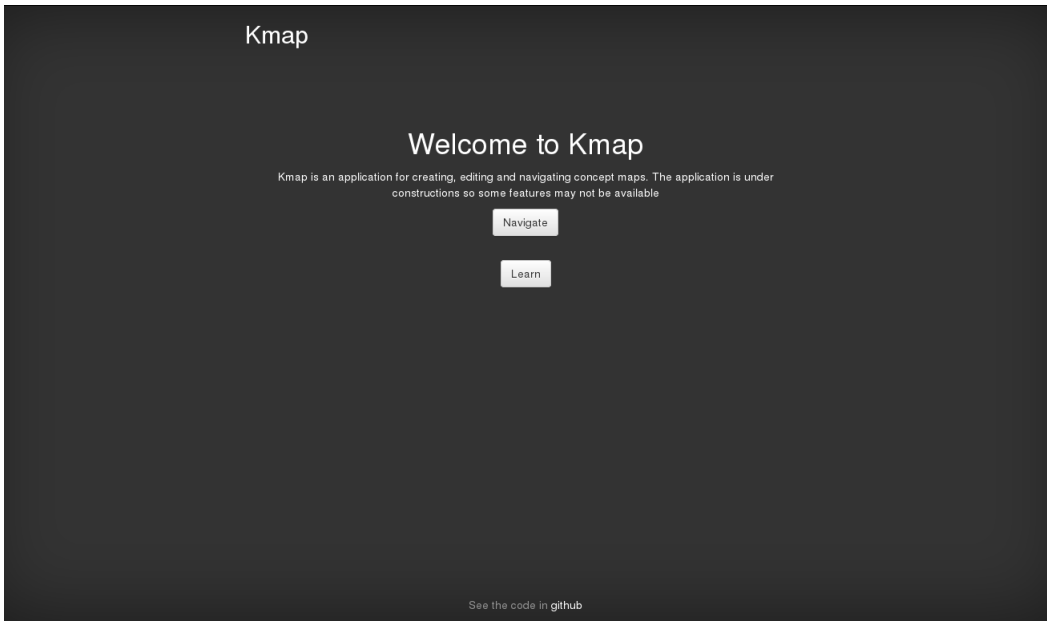

Figura 11: Pantalla de inicio

Si se pulsa en el botón de *navigate* se llega a la pantalla 12, en ella se muestra la funcionalidad que se está ejecutando, en este caso *navigate*. La mayor parte de la pantalla la ocupa el *playground* o área de trabajo, que de momento se encuentra ocupada por un cuadro de dialogo que insta a buscar un concepto para empezar a usar la aplicación. Al pulsar el botón de búsqueda se realiza una petición *GET* a la *API* con el identificador especificado e.j. *GET concepts/concept*.

Si al introducir un concepto este no se encuentra en la base de datos, la a*API* responderá con un código de error 404 y se muestra al usuario un mensaje informativo como se observa en la figura 13

Si el concepto que se ha introducido se encuentra en la base de datos, se realiza la petición de obtención de sus vecino a través del comando *GET /concepts/?neighbor=id* y la aplicación mostrará el concepto buscado, así como sus vecinos más cercanos. En la figura 14 se puede observar un ejemplo. El concepto central o raíz tiende a ser colocado en el medio del área de visualización, pero ademas para hacer más sencilla su identificación se muestra de un color más claro que el de los demás conceptos.

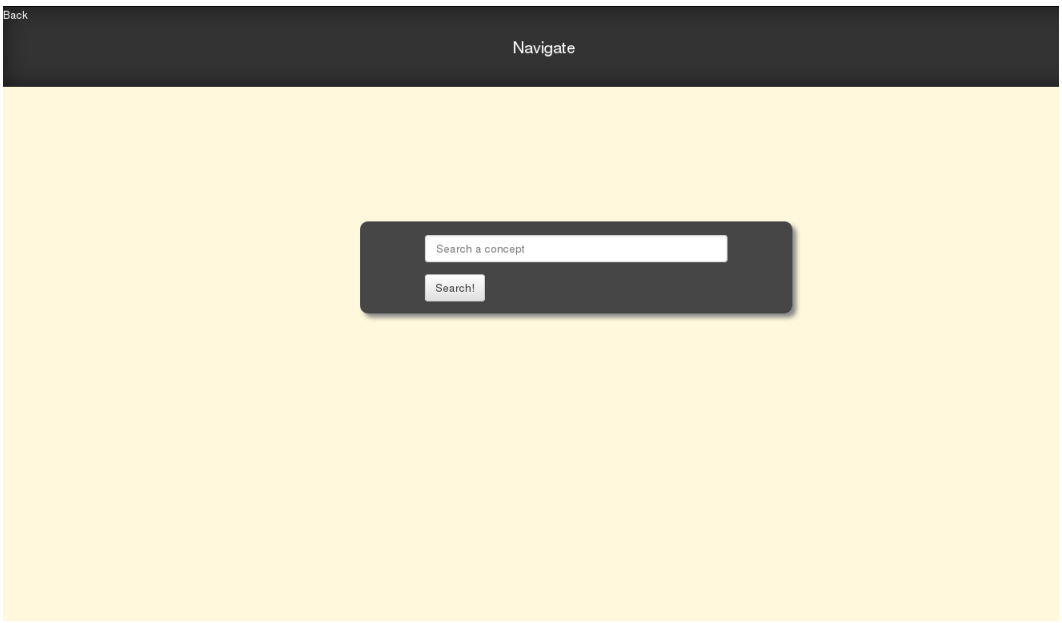

Figura 12: Pantalla *navigate*

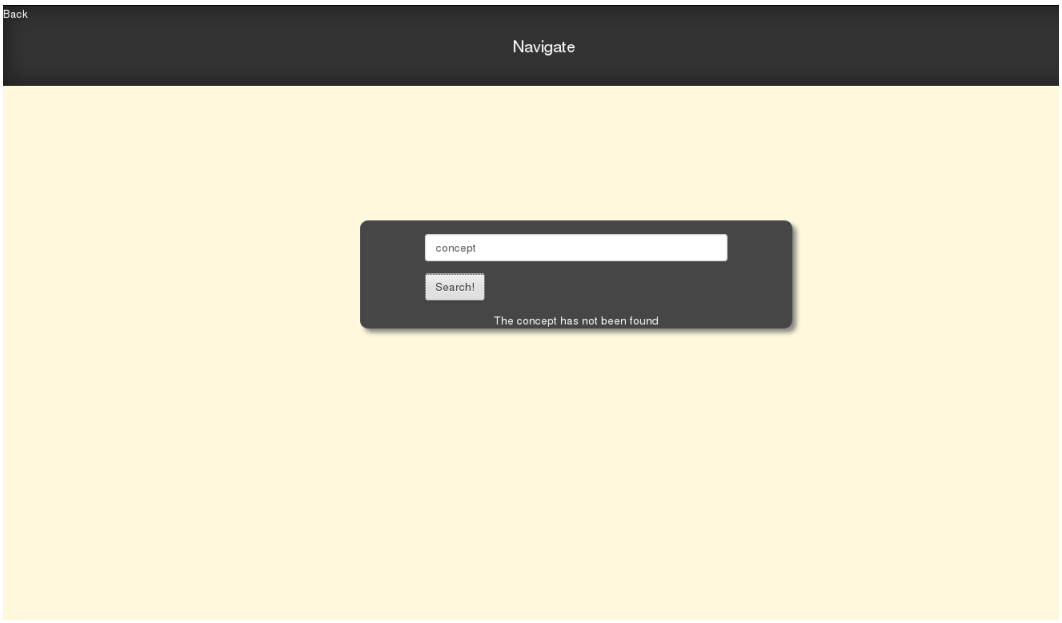

Figura 13: Mensaje de error al no encontrar un concepto

El mecanismo de esta página consiste en buscar un concepto y luego ir navegando por el mapa, explorando las relaciones entre los conceptos. esto se consigue haciendo clic en uno de los conceptos que aparecen rodeando al nodo central. El concepto seleccionado pasa a ser el nuevo nodo central y se cargan los correspondientes vecinos.

Al trabajar con Conceptnet surgió la problemática de nombres o etiquetas de conceptos relativamente largos, esto se ha solucionado generando el tamaño de los nodos a mostrar

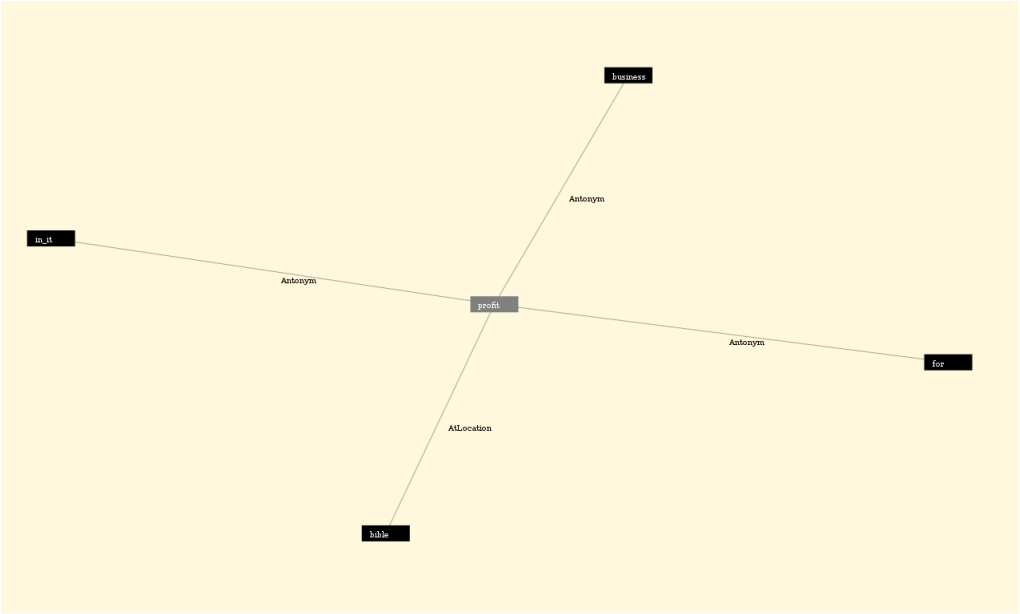

Figura 14: Ejemplo de un concepto y sus vecinos

de forma dinámica. En la figura 15 se puede observar el ejemplo de *holy word of god*, en la que el tamaño del nodo se ajusta al tamaño del concepto.

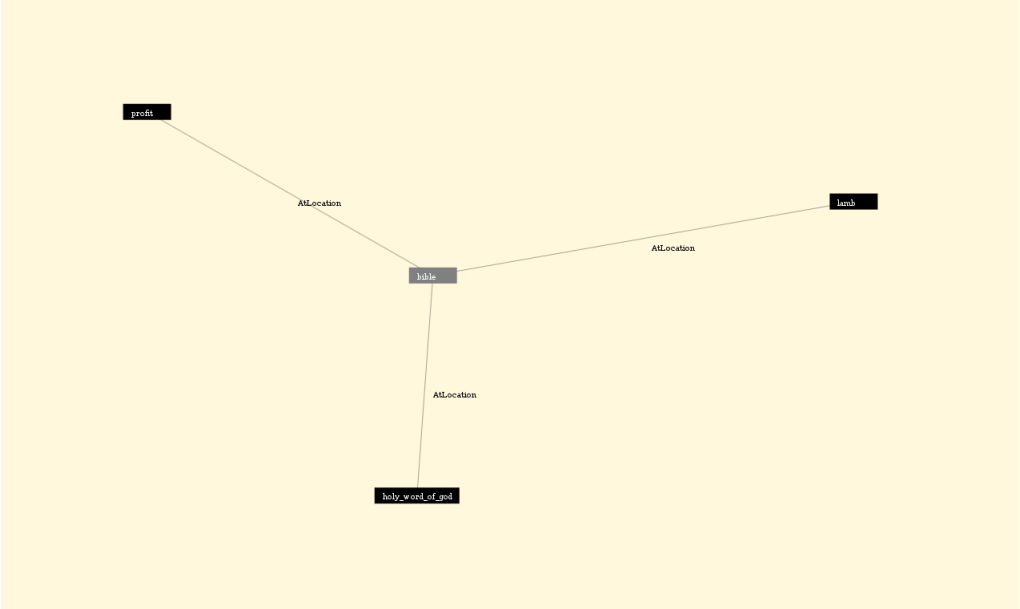

Figura 15: Ejemplo de un concepto con una etiqueta más larga

4. Desarrollo y resultados

4.1. Desarrollo software

4.1.1. Metodología de desarrollo

El método utilizado en el proyecto ha sido el método tradicional en cascada, con tres hitos a completar:

1. *API REST*
2. Prototipo funcional de la interfaz
3. Interfaz web usable

El análisis de del problema, así como el análisis de las posibles tecnologías a utilizar consumió gran parte del tiempo del proyecto, se optó para las primeras pruebas por un *framework* que resultó incompatible con el resto de los módulos del proyecto. Una vez escogidos las diferentes partes que iban a componer la aplicación se tuvo que realizar un análisis del funcionamiento interno de las mismas, dado que al trabajar con una base de datos basada en grafo, se han forzado los límites de dos de los módulos del proyecto, tastpie y Django. Asimismo, el módulo neo4django se encuentra en desarrollo y está escasamente documentado, lo cual ha obligado también ha hacer un análisis del código del mismo.

Para el diseño de la *API REST* se tuvieron reuniones en las que se estableció que funcionalidades básicas debía poseer la aplicación. En cuanto al diseño de clases el proyecto se encontraba acotado por la estructura que requería Django en el lado del servidor y arbor.js en el lado del cliente.

Durante este proceso se tuvieron que hacer cambios en la estructura de las clases, primero, cuando se tuvieron que introducir los enlaces entre los nodos como recurso de primera categoría, pasando a estar modelados como nodos en la base de datos los enlaces de la aplicación, este cambió se realizó para poder dotar a los enlaces de características propias tales como tipo o peso.

Otro de diseño que se realizó a mitad del proyecto fue la re-estructuración de la parte cliente en clases para aumentar la modularidad y poder ser reutilizada en posibles ampliaciones de la aplicación.

Durante la implementación se ha utilizado para la gestión de versiones Git que es un sistema de control de versiones distribuido y de código abierto, y se ha alojado el código en *github* [12]

4.1.2. Métricas software

Las métricas software proporcionan una serie de datos que permiten hacer una aproximación a la calidad y complejidad del código implementado.

[12] `https://github.com/daniels-mer/kmap/tree/master/kmap`

Pylint es una herramienta de análisis de código, tanto a nivel de detección de errores, cumpliendo el papel que corresponderías un compilador en un lenguaje compilado, como a nivel de estándares de código y de diseño de clases. Debido a la naturaleza dinámica de Python el análisis de errores es limitado, pero permite detectar ciertos errores con facilidad, *Pylint* califica con una puntuación entre 10 a teóricamente $-\infty$. Este proyecto ha obtenido una calificación de 7.3.

A continuación se expone la salida de la herramienta:

```
************* Module kmap
C:  1,0: Missing docstring
************* Module kmap.tests
C: 11,0:SimpleTest: Missing docstring
R: 11,0:SimpleTest: Too many public methods (65/20)
************* Module kmap.api
C: 90,0: Line too long (90/80)
C:115,0: Line too long (90/80)
C:116,0: Line too long (116/80)
C:256,0: Line too long (90/80)
C:289,0: Line too long (81/80)
C:292,0: Line too long (81/80)
W: 29,0:ConceptResource: Method 'obj_delete_list_for_update' is abstract in
class 'Resource' but is not overridden
W: 29,0:ConceptResource: Method 'apply_filters' is abstract in class 'Resource'
but is not overridden
C: 29,0:ConceptResource: Missing docstring
W: 41,4:ConceptResource.Meta: Class has no __init__ method
C: 41,4:ConceptResource.Meta: Missing docstring
R: 41,4:ConceptResource.Meta: Too few public methods (0/2)
R: 29,0:ConceptResource: Too many public methods (77/20)
W:181,0:LinkResource: Method 'obj_delete_list_for_update' is abstract in class
'Resource' but is not overridden
W:181,0:LinkResource: Method 'apply_filters' is abstract in class 'Resource'
but is not overridden
C:181,0:LinkResource: Missing docstring
W:192,4:LinkResource.Meta: Class has no __init__ method
C:192,4:LinkResource.Meta: Missing docstring
R:192,4:LinkResource.Meta: Too few public methods (0/2)
W:274,15:LinkResource.obj_create: Catching too general exception Exception
R:181,0:LinkResource: Too many public methods (77/20)
************* Module kmap.models
C: 30,4:Concept.nodelinks: Missing docstring
E: 32,58:Concept.nodelinks: Instance of 'Concept' has no 'links' member
E: 35,58:Concept.nodelinks: Instance of 'Concept' has no 'links' member
C: 52,4:Link.opposite: Missing docstring
E: 53,11:Link.opposite: Instance of 'Relationship' has no 'all' member
E: 54,19:Link.opposite: Instance of 'Relationship' has no 'all' member
E: 56,19:Link.opposite: Instance of 'Relationship' has no 'all' member
```

```
E: 59,11:Link.__str__: Instance of 'Relationship' has no 'all' member
E: 60,38:Link.__str__: Instance of 'Relationship' has no 'all' member
E: 61,35:Link.__str__: Instance of 'Relationship' has no 'all' member
************* Module kmap.urls
C: 23,0: Invalid name "concept_resource" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C: 25,0: Invalid name "v1_api" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C: 29,0: Invalid name "urlpatterns" (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)


Report
======
256 statements analysed.

External dependencies
---------------------
::

    django
      \-conf
      | \-urls
      |   \-include (kmap.urls)
      |   \-patterns (kmap.urls)
      |   \-url (kmap.urls)
      \-core
      | \-exceptions
      | | \-ObjectDoesNotExist (kmap.api)
      | \-serializers
      |   \-json
      |     \-DjangoJSONEncoder (kmap.serializers)
      \-shortcuts
      | \-render (kmap.views)
      \-test
        \-TestCase (kmap.tests)
    neo4django
      \-db
        \-models (kmap.models)
    neo4jrestclient
      \-client
        \-GraphDatabase (kmap.api)
        \-Node (kmap.api)
    tastypie
      \-api
      | \-Api (kmap.urls)
      \-authorization
      | \-Authorization (kmap.api)
      \-bundle
      | \-Bundle (kmap.api)
```

```
      \-exceptions
      | \-BadRequest (kmap.api)
      | \-NotFound (kmap.api)
      \-fields (kmap.api)
      \-resources
      | \-Resource (kmap.api)
      \-serializers
        \-Serializer (kmap.serializers)
```

Raw metrics
----

type	number	%	previous	difference
code	285	50.00	285	=
docstring	223	39.12	223	=
comment	14	2.46	14	=
empty	48	8.42	49	-1.00

Messages by category
----

type	number	previous	difference
convention	17	17	=
refactor	5	5	=
warning	7	7	=
error	8	9	-1.00

% errors / warnings by module

```
+------------+-------+--------+---------+-----------+
|module      |error  |warning |refactor |convention |
+============+=======+========+=========+===========+
|kmap.models |100.00 |0.00    |0.00     |11.76      |
+------------+-------+--------+---------+-----------+
|kmap.api    |0.00   |100.00  |80.00    |58.82      |
+------------+-------+--------+---------+-----------+
```

```
Messages
--------
```

```
+-----------+------------+
|message id |occurrences |
+===========+============+
|E1101      |8           |
+-----------+------------+
|C0111      |8           |
+-----------+------------+
|C0301      |6           |
+-----------+------------+
|W0223      |4           |
+-----------+------------+
|R0904      |3           |
+-----------+------------+
|C0103      |3           |
+-----------+------------+
|W0232      |2           |
+-----------+------------+
|R0903      |2           |
+-----------+------------+
|W0703      |1           |
+-----------+------------+
```

```
Global evaluation
-----------------
Your code has been rated at 7.30/10

Duplication
-----------

+-------------------------+------+---------+-----------+
|                         |now   |previous |difference |
```

nb duplicated lines	0	0	=	
percent duplicated lines	0.000	0.000	=	

Statistics by type

type	number	old number	difference	%documented	%badname
module	7	7	=	85.71	0.00
class	9	9	=	44.44	0.00
method	29	29	=	93.10	0.00
function	2	2	=	100.00	0.00

5. Conclusiones y trabajo futuro

5.1. Conclusiones

Durante este trabajo se ha desarrollado un servicio web, que permite utilizar mapas de conceptos, tanto desde una interfaz web, como a traves de una *API REST*. Los objetivos del trabajo se han visto cumplidos de forma parcial, la *API REST* se encuentra completamente operativa, con todos los comandos necesarios funcionando, así como aceptando algunos parámetros para comportamientos personalizados. Por otros lado la interfaz web no ha sido completada de forma satisfactoria, llegándose a producir solamente la aplicación que permite navegar a través de un mapa ya creado con anterioridad, y faltando dos de las aplicaciones que se habían planteado en un principio, los ejercicios, y la aplicación de creación de mapas propios.

Sobre el estado final de la aplicación se puede concluir que si bien presenta puntos de mejora, muestra aun potencial increíble. Además, el esfuerzo en diseñar un sistema modular y de capas permitiría incrementar el número de funcionalidades con el mínimo esfuerzo de integración.

A nivel de desarrollo personal, este trabajo ha servido para ganar un inscribible conocimiento del funcionamiento de las aplicaciones *REST*, así como de el desarrollo de módulos de Python de un nivel de complejidad superior al que había trabajado hasta entonces. Así mismo, la programación JavaScript que se ha desarrollado se ha alejado de la que estaba habituado, acercándose también a un proyecto estructurado, más que unos simples *scripts* de manejo dinámico del *DOM*

5.2. Trabajo futuro

Se han identificado numerosos aspectos susceptibles de mejora en la aplicación. Tanto a nivel de implementación como de adición de nuevas funcionalidades.

Primero de todo, se debe hacer un estudio sobre la viabilidad de seguir apoyándose en neo4django, dado que este modulo es el que limita de forma más notable el desarrollo de la aplicación. Así pues, se ofrecen varias posibilidades en el futuro, una de ellas es contribuir de forma activa con el proyecto neo4django, beneficiándose así de las mejoras que este vaya produciendo. Otra de las posibilidades es desechar por completo el uso de este módulo, trabajando directamente con la *API* de Neo4j, teniendo que re escribir parte del código existente, pero eliminando algunas de las susodichas limitaciones. La última opción es seguir con el modelo actual, manteniendo el módulo neo4django actualizado e intentar sortear los escollos que produce.

Por parte del lado del servidor, existen 2 funcionalidades relacionadas con tastypie que harían la aplicación más robusta, manejo de usuarios y de forma ligada a este un sistema de autentifican y autorización de permisos de los usuarios sobre los recursos. Finalmente, por parte del servidor, la funcionalidad de operar con distintos mapas de conceptos, las últimas versiones de Neo4j están incorporando el manejo de subgrafos de forma nativa, pero si fuera imposible actualizarse a la última versión por temas de compatibilidad, se podría realizar a través de los indices de Neo4j.

Por último, en la parte de cliente querían pendientes como funcionalidades deseadas, l página e ejercicios didácticos, en la que presentar un mapa parcial que el usuario tiene que completar. Y una página desde donde se puedan crear o modificar mapas de conceptos por los usuarios, dado que la única forma posible actualmente es a través de la *API*.

Referencias

[1] D. Novak, Joseph Concept mapping: A useful tool for science education. Article first published online: 3 FEB 2011

[2] Mallari Mistades, Voltaire Concept Mapping in Introductory Physics. Journal of education and human development

[3] Scofield, Ben NoSQL - Death to Relational Databases(?). Fecha de consulta[26 Junio 2014] Disponible en http://www.slideshare.net/bscofield/nosql-codemash-2010

[4] Ian Robinson, Jim Webber y Emil Eifren. *Graph Databases* page 20, 1st ed. O'really 2013

[5] Robert Speer y Catherine Havasi. *Representing General Relational Knowledge in ConceptNet 5*

[6] Call diagram of django tastypie resource get_list method. Fecha de consulta[10 de Mayo de 2014] Disponible en http://codeslashslashcomment.com/2012/11/13/call-diagram-of-django-tastypie-resource-get

[7] Fielding, Roy T.; Taylor, Richard N. (May 2002),*Principled Design of the Modern Web Architecture*

[8] Cubo Velázquez, Alberto Representational State Transfer (REST). Un estilo de arquitectura para Servicios Web. Panorámica y estado del arte.